PROGRAMMING YOUR WAY OUT OF THE PAST

ISIS and the META Project

KENNETH BIRMAN
KEITH MARZULLO

Basic changes in the way people use and program distributed systems are on the horizon. These changes range from new low-level programming techniques that make it easy to build fault-tolerant distributed applications that exploit replication and concurrent execution to a *meta-operating system* with services spanning large numbers of machines in heterogeneous networks.

A Programming Revolution

Users of distributed-computing systems rapidly discover how similar such systems are to the time-shared machines of the 1970s: The pervasive use of "network transparency" techniques largely conceals the fact of distribution. For example, the

dominant distributed-programming technology, *remote procedure call* (RPC), permits a program running on one machine to invoke a procedure residing in some other program. Given adequate language support, an RPC interface can hide many details of message-based interaction and connection management from a user. The idea of transparency also extends to other parts of a typical distributed system. Using a filesystem such as NFS, a program can operate on files that physically reside on a remote machine in the network using exactly the same interface as for local files. If the distributed-computing revolution is underway, its impact on how programs are written has been minor.

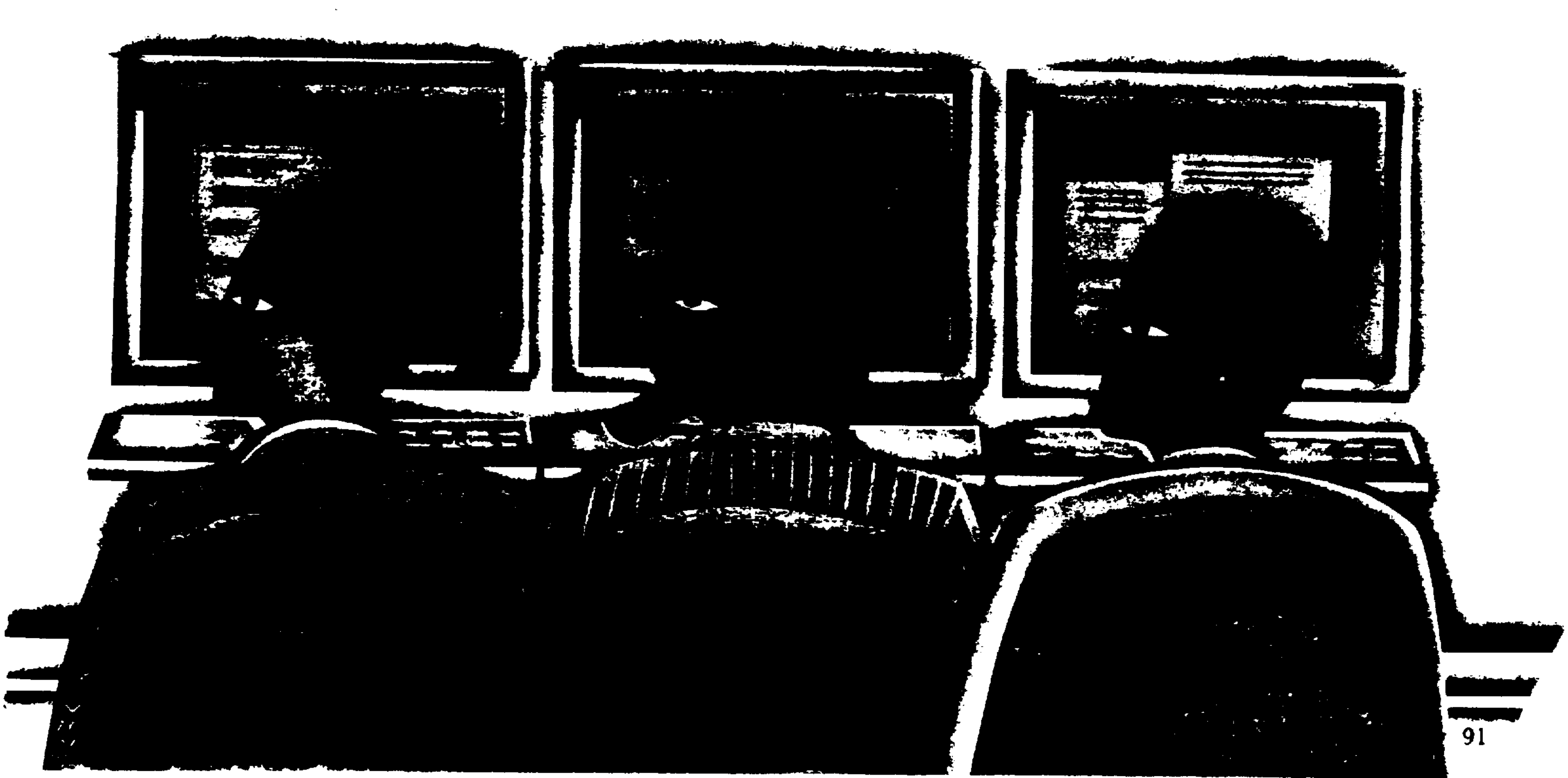
This lack of impact is troubling, especially in light of the many reasons why distributed computing *should* be different from nondistributed programming. Parallel computing is certainly analogous to distributed computing in many respects. Yet, whereas the effective use of parallel machines has lead to fundamentally new programming technologies such as Linda[1] and CSP,[2] the same thing has not happened in the case of distributed programs. If distributed systems are built using technologies that proved to be unsatisfactory in parallel settings, then distributed systems are probably making ineffective use of concurrency (parallelism).

The requirements of a distributed application may go *beyond* those of a timeshared or parallel program. Not only does a distributed computer system need to exploit concurrency, but it may also need to remain operational in the presence of "partial" failures—that is, situations in which one of the machines connected to a network fails or becomes partitioned from the others, while the majority of the machines remain operational and must reconfigure themselves and continue executing. The complementary problem of reintegrating a recovered machine into an online system also arises.

From this perspective, transparency may not be such a tremendous win. RPC is a *pairwise* programming methodology: although RPCs can nest, more complex interactions are not normally constructed from RPCs. A transparent RPC mechanism offers little in applications that require concurrent action by more than two processes at a time, especially if those processes must cooperate but are not controlled by a common ancestor. Moreover, most RPC mechanisms handle failures by either timing out or by retrying a request several times, at best providing some form of "at most once" guarantees. This is not a sophisticated way of reacting to a failure.

The problem is that the gap between these mechanisms and a coordinated algorithm (where-

Illustration: *Jack Desrocher*



by a set of processes joins forces to solve a problem in a fault-tolerant manner) is simply too large for the average programmer to bridge. The unfortunate user whose distributed problem doesn't fit into these paradigms must undertake a complex and costly system-development effort or abandon a distributed solution entirely.

One approach to these problems is to augment RPC with *transactional* features oriented toward controlling concurrency and ensuring that persistent objects can recover their states after failures. Prominent among efforts to do this are the Argus system, which focuses on language issues,[3] and the Camelot system, which focuses on performance.[4] These sorts of "better behaved" RPC mechanisms will doubtless play a major role in the distributed systems of the future. On the other hand, they are not resulting in fundamentally new strategies for exploiting the network, and after a 10-year development period, their most important role has been in creating and managing special-purpose databases.

The major premise of the authors' project, ISIS, is that when a distributed system is viewed as a timeshared system or encourages its users to program as if their application were running on a dedicated idle system, as with transactional RPC, the most powerful resource that a distributed system offers us is lost: distribution itself. We lose the ability to employ a set of processes in a coordinated, cooperative attack on a problem. We lose the ability to apply highly adaptive, reconfigurable solutions to applications that must remain on line in the presense of failures and recoveries. And, we lose the possibility of building a distributed system that is more fault tolerant and offers higher performance than any of its components.

As we enter the 1990s, the state of the art in distributed computing embodies a paradox. On the one hand, networks are becoming ubiquitous and can be used and programmed much like the timeshared processors of the 1960s and 1970s. On the other hand, the proliferation of networks has yet to result in any sweeping changes in the way we develop software. Furthermore, a large class of applications seems to lie out of reach: those that require direct coordination among a set of processes, replicated data, parallel execution of requests, or a coordinated response to a failure or some other reconfiguration event. The techniques that have helped achieve networked computing offer no easy solutions to these *intrinsically distributed* applications.

A new programming technology now promises to open the door to solving these applications: the ISIS[5] Programming Toolkit. The ISIS Toolkit is a "low level" programming technology. It can change the way distributed systems are built, but it will not directly change the available higher-level services of distributed operating systems.

Concurrent with the ISIS Toolkit is the META project, which is reexamining high-level mechanisms taken for granted in distributed systems—the filesystem, the shell language, and the administration tools.

ISIS and META, together with other technologies, herald basic changes in the way we think about and use computer networks. Rather than viewing networks primarily as a way to connect a program running in one place with a resource that "lives" someplace else, these technologies permit distributed software design that makes explicit use of the distributed character of the network environment. Although no one system addresses the whole spectrum of distributed requirements, taken as a composite they offer a sweeping range of new and powerful ways to approach distributed problems.

The ISIS Distributed-Computing Model

Prior to a detailed review of the ISIS toolkit, an understanding of some of the programming structures of ISIS is helpful. Like most distributed systems, ISIS is based on processes and messages. Our notion of a process is the basic UNIX one: each execution of a program gives rise to a process—an address space containing one or more lightweight tasks (also called threads of control or lightweight processes). In ISIS, each arriving message is handled by a separate task. Although task execution is FIFO and nonpreemptive, tasks can explicitly wait on and signal *condition* variables when desired. (For more on lightweight tasks, refer to *The ISIS Programming Manual and User's Guide*.[6])

ISIS assumes that both processes and the communication system can fail. ISIS is limited in the kinds of failures it can handle. It tolerates *communication failures* that involve lost messages, but it may hang if communication is completely disrupted between sets of sites by a network partitioning.[7]

ISIS also supports reconfiguration and continued execution after crash failures, whereby programs or machines simply stop executing (most software failures result in crashes of this sort). It is of limited value in a system subject to more extreme crash modes, such as when a software bug causes a program to go into an infinite loop or to send messages containing incorrect data. Because ISIS has no way to distinguish these behaviors from correct ones, neither condition can be detected. Yet, if the application designer can detect such a condition, ISIS does offer tools to overcome it.

Communication among ISIS processes is by messages. These contain streams of typed data items, which are added to the message by use of format strings (shown below). Because ISIS knows the type of each data item, the reception side

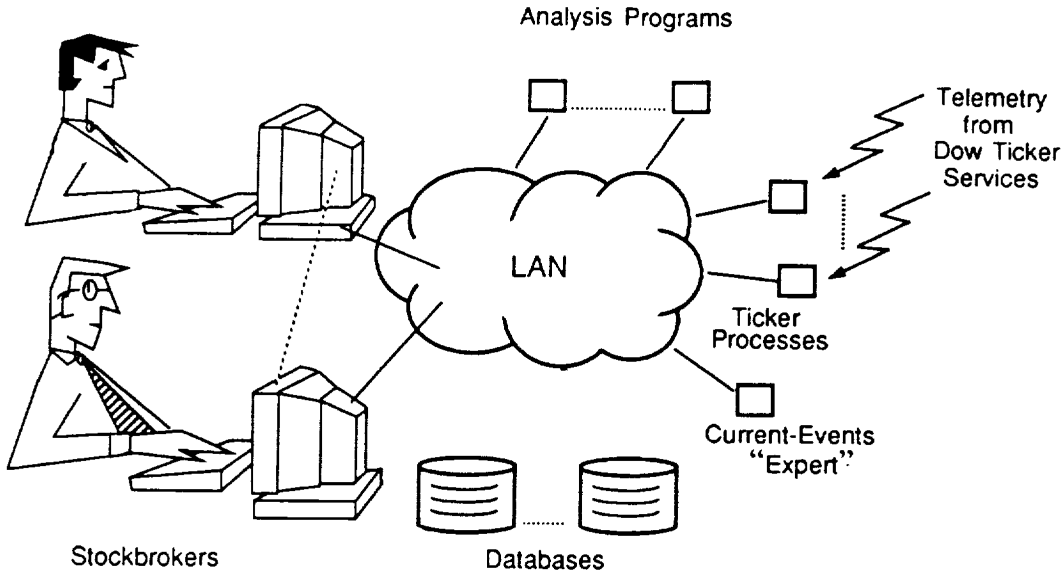


Figure 1. *A stockbroker's trading system*

handles byte-order conversions automatically.

ISIS also supports *virtually synchronous process groups*. These groups consist of a set of processes that are cooperating for "some purpose," be it distributed execution of requests, management of replicated data, or whatever. A process can belong to many process groups, and the members of any particular group need not be identical or even be programmed in the same language, although they are expected to use compatible group-management algorithms. Each group has a symbolic pathname and a unique 24-byte address that can be used to communicate with it. Group addresses and process addresses can be used interchangeably throughout ISIS; when a message is sent to a group address, the system expands this into a *reliable broadcast*[8] to the current membership of the group.

Before saying more about the notion of virtual synchrony, or even what it means for a broadcast to be reliable, we look at some typical ways that ISIS applications use process groups. To make the example concrete, consider a stockbroker's trading system composed of three types of entities (see Figure 1). At the front end, the system has workstations that display current quotes and trading advice. These employ an interactive command interface. Connected to the system are "ticker" devices, the computer-readable analog to the mechanical stock tickers used in the past. In this system, tickers are redundant because the risk of a failure must be kept to a minimum. (For simplicity, the process that handles a given ticker device is referred to as a ticker.)

The system also provides a variety of analysis services capable of searching databases for information needed by the trader, calculating suggested buy and sell margins based on trend analyses, comparing options and futures prices with current quotes, and other tasks.

A system such as this could use ISIS process groups in several ways. At the front end, the set of stocks that any given broker is monitoring will likely vary over time, perhaps quickly in modern program trading. If a ticker process receives a new quote for Sun, how is it to know what workstations currently need this information?

An easy solution is to create a process group for each stock currently being monitored. All programs wanting quotes for that stock would join the group. A ticker would then broadcast quotes to the appropriate process groups and leave ISIS to cope with their dynamically changing membership (see Figure 2).

Remember that for reasons of fault tolerance, ticker processes are redundant. A failure might be due to the crash of a ticker, or it might be due to a "softer" problem such as a transient overload or a burst of line noise that garbles a quote. Ticker processes can also be redundant for purposes of load sharing. Even in the absence of failures, a single ticker cannot practically deal with all quotes on behalf of all brokers in a large trading room. Forming the ticker processes into a process group makes programming the necessary control algorithm easy.

ISIS permits users to assign responsibility based on the initial letter of a stock's name in a manner

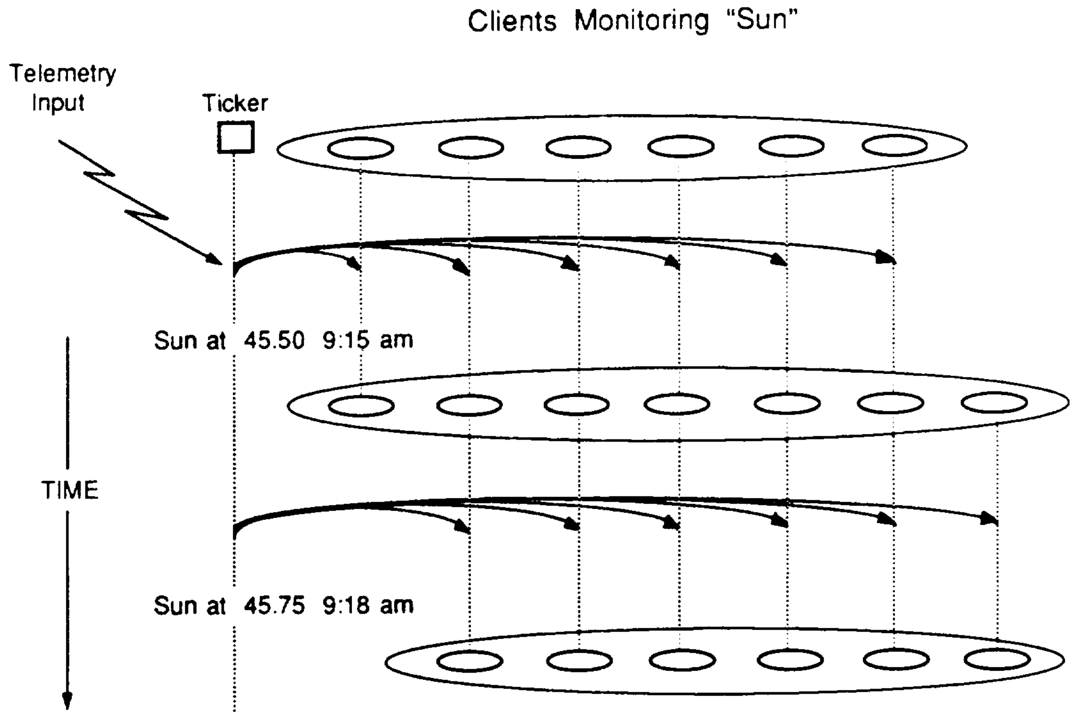


Figure 2. *Tickers must deal with a dynamically changing clientele*

that rules out confusion about which ticker handles which stocks—that is, there is no risk of having two tickers watching stocks starting with the letter A and none watching stocks starting with the letter C. After a system crash or recovery, ISIS provides a reconfiguration method—in this case, reassigning the letters to ensure full coverage. It also provides a way to cope with tickers that garble or miss individual quotes and to retransmit quotes that arrive between the time when a ticker fails and when reconfiguration takes place. One could implement a redundant assignment rule—for example, by arranging for two tickers to handle each stock, as shown in Figure 3. Although quotes arrive in duplicate and stale quotes must be discarded, when a failure occurs, quotes keep flowing. Two nearly simultaneous failures would have to occur to prevent a broker from obtaining timely information.

The same techniques that can be used to assign stocks to tickers can also be used to subdivide other types of computations. For example, a complex database search can be divided into parts that each member of a set of servers will perform independently (merging the results at the end). The same mechanisms can also be used for an analysis that requires opinions from multiple "experts."

Another area in which ISIS offers sophisticated support is in the use of replicated data. Process group members can easily maintain replicated data structures, updating them at extremely low cost and permitting direct read access, much as with an accurate cache.

Replicated data is an example of a group state that changes dynamically. Many servers need to maintain dynamic state information, be it replicated data, descriptions of pending requests, or lists of currently held locks. When a process recovers and needs to join an operational system, transferring this information poses a thorny problem. Because the information changes in response to some types of events, one must lock out those kinds of events while copying the state of the group to the new member. Clearly, the transfer must be fault tolerant.

Overall, these factors add up to the kind of problem most programmers would find hard to solve. ISIS, however, has a group-join mechanism that automates the task. The programmer supplies a routine to transfer the group state out to the new member and a routine to receive and unpack the state when it arrives. ISIS correctly synchronizes the "join" and handling failures. Clients using the group will usually be unaware that a new member joined while they were talking to the group. The method works well for states of up to a few hundred kilobytes in size, which is enough for most database purposes (additional mechanisms for groups managing much larger states are now being designed).

What would ISIS *not* be useful for? One major area is transactional database and file management.[9] Some powerful systems for this task are

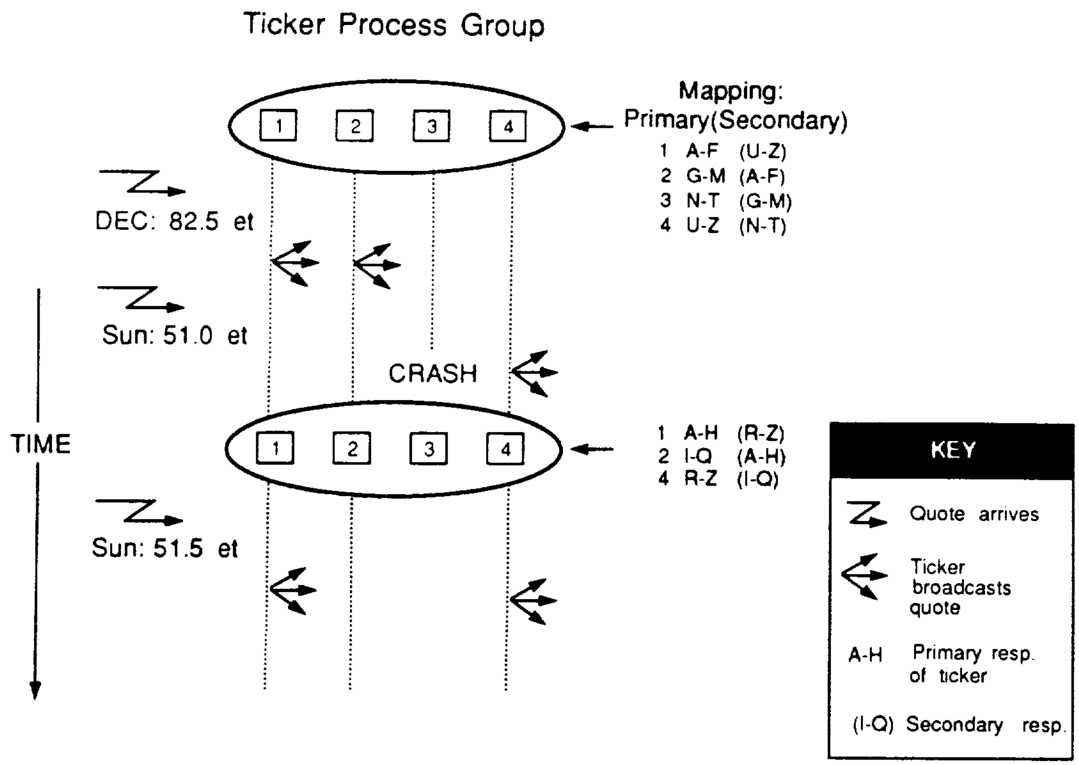


Figure 3. *Using redundancy to obtain fault-tolerant quote dissemination.*

available, and ISIS was designed to avoid duplicating these efforts. As a result, although ISIS has powerful synchronization mechanisms, it is not oriented toward serialization (a widely accepted technique for maintaining database correctness) or atomic transactions (the usual technique for database crash recovery). Instead, ISIS focuses on cooperative distributed algorithms and on the volatile and rapidly changing state of a distributed computation.

Virtual Synchrony

ISIS is not the first system to use process groups, but its process-group mechanism is unusually powerful. The reason for this power is a theoretical advance called *virtual synchrony*.

When reading about the various schemes for subdividing work among a set of tickers or a collection of expert subsystems, you may have wondered how these schemes can possibly be correct when failures and recoveries occur. Certainly, almost any algorithm that uses RPCs for interactions between the ticker processes and time-outs to detect failures will be complex and prone to errors. The risk of ending up in a state in which no process sends quotes for IBM stock seems very real (for example, one process covers A-H, another J-N, but no process covers I). This situation occurs if processes have inconsistent views of one another's status. And, such an inconsistency can easily arise because transient phenomena and overloads can mimic failures in networks of workstations. Worse, failures and other events might be observed in different orders by different processes in a distributed system. One can imagine an algorithm that behaves differently on detecting a given event in one state than in another, and that changes state in response to events it observes. Two instances of this algorithm might not give the same behavior even when executed on the *same* events but in different orders. If one treated a transient overload as a failure but the other did not, inconsistency would certainly arise.

To take an extreme example from a different setting, consider a factory that produces some sort of chemical and that the production strategy changes if some critical valve is not responsive. A programmer might decide to decentralize control among a set of control programs to gain increased fault tolerance and benefit from load sharing. Should the programmer not now be worried that one component of the control subsystem could *incorrectly* conclude that the valve has jammed, perhaps because of a communication failure (and hence switch to the emergency shutdown procedure), while the remainder of the system continues normal operation, unaware of what has happened? Clearly, correct behavior in each of the components of a system does not automatically imply mutually correct behavior of the system as a whole.

In light of these sorts of problems, how can one be sure that a rule like the one we proposed for controlling the set of tickers will operate correctly

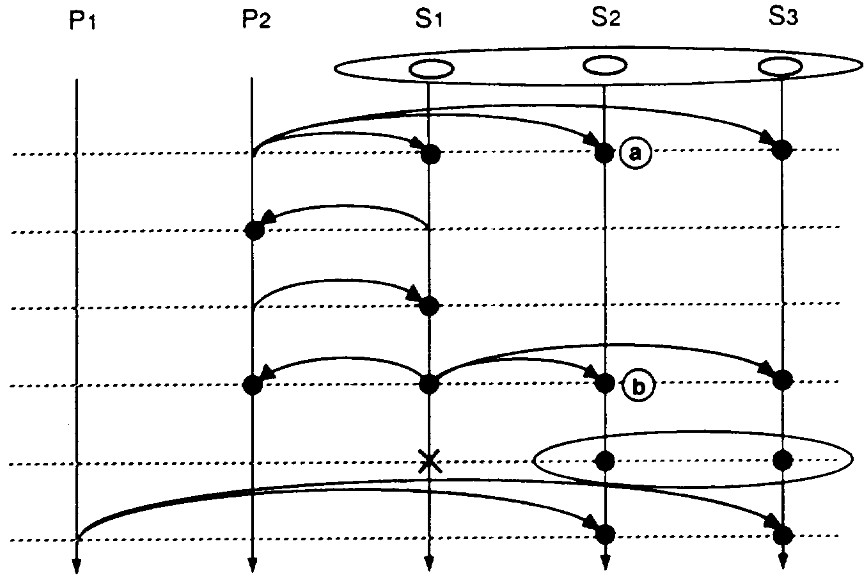
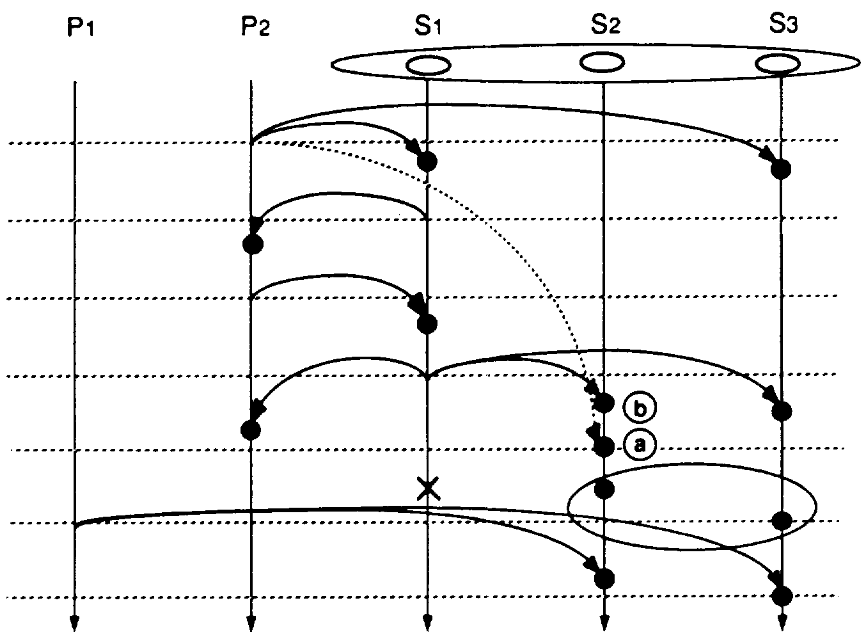


Figure 4. *In a synchronous execution, only one event can happen at a time. However, a single event can occur (or be observed) at multiple places, as with the delivery of a broadcast or the detection of a failure.*

Figure 5: *A virtually synchronous execution is indistinguishable to the application from a synchronous one. Event orderings may differ from process to process but only if behavior of the application is unaffected.*



in other applications? How is it possible to decentralize a factory-control system with confidence that it will still function correctly?

Virtual synchrony efficiently solves this class of problems. It starts with a simplified model for how a distributed system executes—the program is coded as if this model were realistic. Then, much as a compiler may produce code that differs from the original program without changing its behavior, the distributed program is executed in ways that improve efficiency and permit it to run in a realistic environment while preserving correctness. Because the program is written for a setting that differs from the one in which it runs, ISIS supports a *virtual* environment.

The idealized environment of ISIS is illustrated by the distribution of quotes to a set of application programs shown in Figure 4. Here, time advances from top to bottom, with one distributed event occurring per time interval. The types of events shown include broadcasts of quotes, failures, and joins. Notice that a broadcast to a group is always delivered to all the current members at once. Similarly, all group members see failures in unison.

A synchronous system can be inherently costly and scale poorly. To avoid this problem, ISIS runs programs designed for synchronous environments in a much more concurrent manner by relaxing any kinds of synchronization that the algorithm doesn't really depend upon. For example, if a synchronous algorithm doesn't look at a realtime clock, it will execute correctly even in a "loosely" synchronous setting, where event *orderings* are the same as for a synchronous environment; but the actual time at which events occur or messages are delivered can differ from process to process. A virtually synchronous environment goes further: in many cases, it presents events in different orders by different processes, provided that they all behave indistinguishably from some synchronous execution (see Figure 5).

The idea of virtual synchrony is rooted in database and distributed systems theory.[10] One reason why the idea was not applied to distributed systems sooner is that it needs a careful analysis of the ordering requirements of the application being run, and lacking such an analysis, performance is certain to be poor. ISIS applications interact through our toolkit, however, so the toolkit algorithms can be analyzed and optimized, and this benefits ISIS applications as a whole.[11]

In practical terms, virtual synchrony means that ISIS applications execute in a simplified environment in which a layer of software hides many of the difficulties that make distributed programming so hard. All the ISIS tools have simple interfaces and simple behavioral descriptions that include failure cases. User code carries little risk of unpleasant surprises such as race conditions, inconsistent views of the status (failed or operational) of processes, or ordering anomalies that can lead to inconsistent behaviors in different parts of a system.

A brief review of the ISIS tools, with code fragments as examples, makes the idea of virtual synchrony more concrete. All ISIS tools can be used from C , Lisp, and FORTRAN and, if desired, in conjunction with other mechanisms such as Suntools or NeWS and X Windows.

Messages

ISIS defines a new type of object called a *message*. ISIS uses messages in various ways. A message is created and manipulated much as an input/output stream is. The sequence in Figure 6 creates a message and then scans the contents into variables *stock, date, time,* and *quote*. Notice the similarity to **fprintf** and **fscanf**. Format items may specify base types (as above), variable-length arrays, or user-defined structures.

```
message *mp;
mp = msg_gen(''%s, %s %d, %f'', ''SUN'', ''3/17/89'', 1022, 162.5);
msg_get(mp, ''%s, %s %d, %f'', &stock, &date, &time, &quote);
```

Figure 6. *Creating a message and scanning the contents into* stock, date, time, *and* quote *variables.*

The most common thing to do with a message is to send it to an entry point defined by another process. Rather than require three steps for this process (generate, transmit, deallocate), many ISIS system calls combine all of these steps into a single action (see below.)

Joining Process Groups and Obtaining Group Views

A process uses the **pg_join** request to join a process group. As Figure 7 shows, several sorts of options can be specified. Here, the calling process requests that it be added to the process group named **/analysis/technology**. The group resides in a group of sites called "New York" (site groups are somewhat like process groups, although less dynamic). The handling of the join depends on whether or not the group is already operational within this group of sites. If it is, the **credentials** string is used to validate permission of the new process to join. A state transfer is then initiated by invocation of the state transfer "out" routine in some operational group member. This encodes the state into one or more messages and then transmits them by calling an ISIS-supplied **xfer_out** routine; for each call, the corresponding "in" routine (here, **gstate_in**) will be invoked remotely. Figure 8 illustrates this.

Figure 7. *Specifying options to join a process group.*

```
gaddr = pg_join(''@NewYork:/analysis/technology'',
    PG_CREDENTIALS, ''signature'',
    PG_XFER, gstate_out, gstate_in,
    PG_MONITOR, gstate_mon,
    PG_INIT, gstate_init,
    PG_LOGGED, TRUE,
    0);
```

The **pg_join** routine behaves differently if the group is not already running. In this case, ISIS creates a new instance of the group from scratch. If the group state is not logged (controlled by **PG_LOGGED**), or if this is the first time the application has ever been started, ISIS calls the **PG_INIT** routine. Otherwise, if the member is one of the last to have failed in the old group, the group state is rolled in from a log automatically maintained by ISIS on behalf of the member. If the log is out of date, the joining process must wait until the last members to fail recover. Logging is not the default and is not likely to be common in ISIS.

Finally, ISIS posts a *monitor*. Each group membership change is reported to the routine **gstate_mon**, and if all group members monitor the membership, the changes are synchronous.

Although *joining* is a multistage algorithm, it looks to the outside world like an instantaneous event (see Figure 9). For example, no broadcast ever reaches some group members before a join, and a broadcast reaches some after, so group members can use the group-membership "view" as part of the algorithm for deciding how to behave. (Remember the ticker example?) This data structure lists the current members in the order they joined. Group membership lists change one

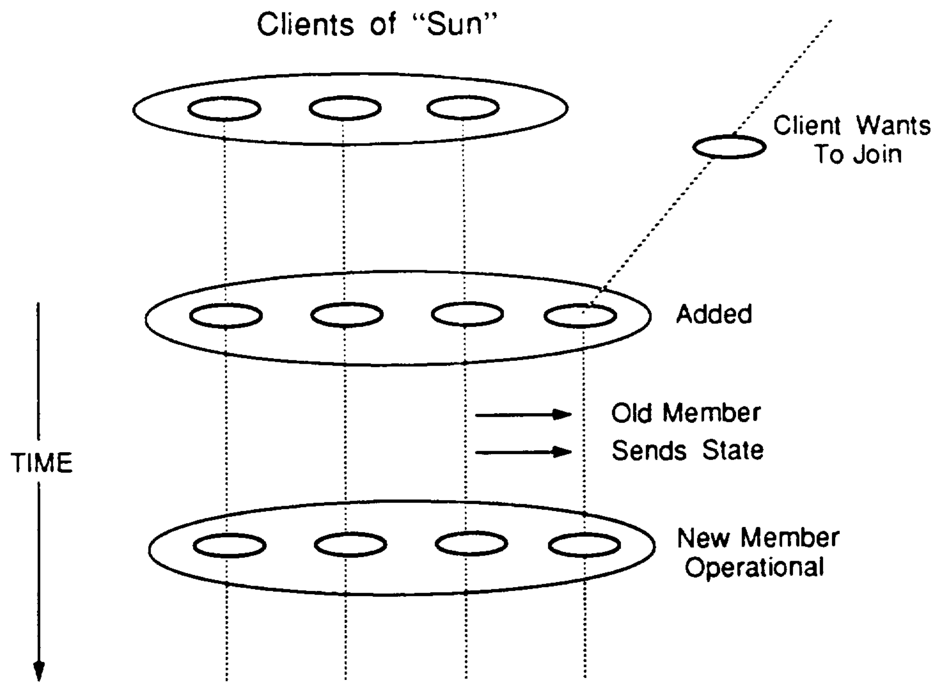
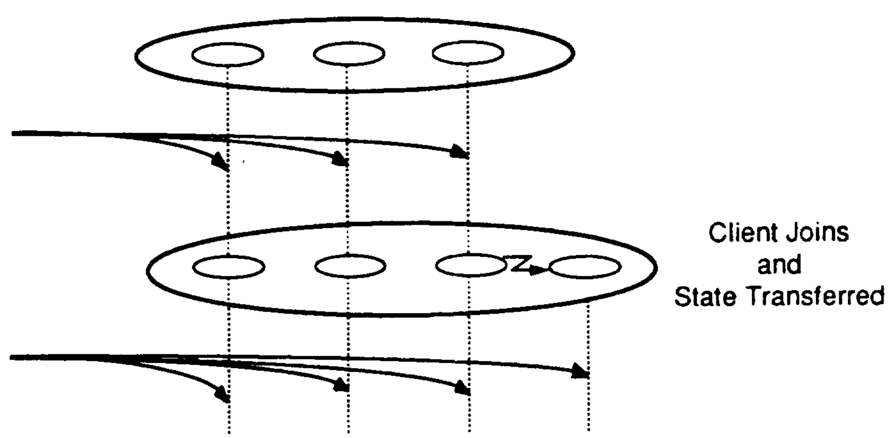


Figure 8. *State transfer to a new member of a process group.*

Figure 9. *Group joins appear instantaneous to the outside world.*



by one, and all members see the same changes and in the same order with respect to other types of events.

Broadcasts and Replies

To broadcast to a process group, a program first looks up its group address and then sends the desired message. If a reply is needed, the caller can wait for one, all, *n*, or a majority of replies. The general format is:

```
bcast(addr, entry, out_fmt, out_data, n_replies,
   input_format, &replies);
```

The sender specifies the address of the destination process or group, the message to send, the number of replies desired, and (if nonzero) the reply format and variables into which the replies should be scanned. Notice that **bcast** combines the interface of **msg_gen** with that of **msg_get**. The ticker process might include code like that in Figure 10, in which the set of clients interested in Sun quotes is represented by a process group and, if the group is not empty, the new quote is transmitted asynchronously (without waiting for delivery to occur). **NEW_QUOTE** is an *entry number*. The idea is that each service exports (as part of its interface definition) the entry numbers of services it provides, much like the procedure identifiers used in an RPC protocol. Users of the service identify their request by specifying an entry number, as shown in Figure 10. Processes that have the service tell ISIS what routine to call when messages of this sort are received. They use:

```
isis_entry(NEW_QUOTE, new_quote, ``got quote'');
```

When the quote arrives, ISIS invokes the designated procedure as a new lightweight task:

```
new_quote(mp)
  message *mp;
  {
     msg_get(mp, ``%s: %s %d, %f'', ....);
  }
```

If needed, a reply can be sent like the original request. Facilities for determining who sent a message, forwarding a message, dealing with failures that occur while waiting for responses, and other instances are available with ISIS.

Replicated Data, Synchronization, and Distributed Computations

ISIS can help a program maintain replicated data, using a broadcast to update it but reading data locally. If desired, a synchronization mechanism based on tokens or locks can be used. Our method is fault tolerant and has a roll-forward recovery scheme when failures occur. More important, it is asynchronous: No process ever blocks when doing a read or an update or releasing a lock (blocking when a lock is *requested* is obviously unavoidable). This means that replicated data in ISIS costs little more than unreplicated data, provided that the network bandwidth can accommodate the background message traffic generated. In practical terms, ISIS V1.2 can update data replicated among 5 processes on separate Sun-3/60 workstations at a rate of 50-100 updates per second.

ISIS has several choices for distributing a computation across the members of a group. A *coordinator-cohort* scheme selects some member to be responsible for a request. Noncoordinator processes function as passive backups, taking no action unless a failure occurs (if replicated data is updated, the coordinator broadcasts its changes).

The approach handles load sharing by permitting multiple coordinators to run simultaneously in different processes when several requests are pending. A *redundant* computation is one in which all members execute a request in parallel, presumably arriving at identical results and changing replicated data in identical ways. A third option is a *subdivided* computation, in which each member performs part of a request, with the collected outcome presented to the caller.

Any of these methods can be programmed with one or two subroutine calls to ISIS. In addition, callers can transmit a request to a subset of the group members, rather than broadcast to all, if only a single response is needed.

Watching and Monitoring

ISIS provides ways to monitor changes to group membership and to watch individual processes for failure. The system ensures that if any process senses a failure, all interested processes will observe the same event. If the failure is transient, the "failed" process will be forced to rejoin the system, discarding or retaining and updating its internal state at the programmer's option.

The toolkit contains other facilities, including an automated program-restart facility that operates after crashes and onsite recovery, a news facility for a program-level analog to the network-news service, and an extensive log-based recovery mechanism. This facility is now being extended to allow an operational group of processes to log information for a process that is inaccessible or down, as an alternative to doing a large state transfer when it recovers.

A Distributed Algorithm for Subdividing a Task

In the ticker example above, recall the problem of subdividing work among a set of tickers. To solve this problem, we can program the tickers to monitor group membership, noting the number of members (*nmembers*) and their relative ranking in the list (*rank*). Given that all members see the same sequence of group views, dividing the alphabet into *nmembers* parts is straightforward, as is assigning responsibility for the *i*th part to the two processes with *ranks* equal to i and $(i+1)$ *mod nmembers*.

Each time membership changes, the work assignment must be recomputed, raising the question of how to synchronize the ticker input stream with the membership changes. Obviously, if the input stream is obtained from the Dow Jones wire service, it will not arrive in the form of ISIS broadcasts. Consequently, if one process switches before some other process does, a gap may result during which coverage of the stocks is incomplete. One way to solve this problem is for processes to operate briefly under two rules simultaneously: the old rule and the new one. Meanwhile, a distinguished process (say, the one with *rank* 0) polls the group to confirm that all members have actually started using the new ranking. When this has occurred, a broadcast is transmitted to inform group members that they can stop using the old ranking. Briefly, clients will have received as many as four copies of some quotes, but none will be missed.

Figure 11 shows a fragment of the corresponding code as it might be implemented in ISIS. Assume that **ticker_mon** was specified in **pg_join**

Figure 10. *Ticker process in which clients interested in Sun quotes are represented by a process group.*

```
new_sun_quote(...)
  {
  address gaddr;
  gaddr = pg_lookup(''@workstations:/stocks/sun'');
  if(!addr_isnull(gaddr))
  bcast(gaddr,NEW_QUOTE,''%s...'',''SUN'',...0);
  }
```

Figure 11. *Program to distribute stock quotes.*

```
#define     POLL     1
#define     SWITCH   2

/* This ticker's primary and secondary responsibilities */
char PrimLow, PrimHigh, SecLow, SecHigh;
/* If non-zero, we are switching to new values */
char OldPrimLow, OldPrimHigh, OldSecLow, OldSecHigh;

#define between(c, low, high) (c >= low && c <= high)

main()
  {
      isis_entry(poll, POLL, ''poll a member'');
      isis_entry(switchover, SWITCH, ''switchover done'');
      pg_join(''@NYC:tickers'', PG_MONITOR, ticker_mon, ...);
      /* Create ''wire monitoring'' task */
      t_fork(watch_wire);
      isis_mainloop();
  }

/* Read quotes from the ticker, as if it were a keyboard */
watch_wire()
  {
     while(TRUE)
     {
         /* Get a new quote from the wire, disseminate */
         read_quote(&stock, &date, &time, &price);
         got_quote(stock, date, time, price);
     }
  }

/* On receiving a new quote, broadcast it if I am
   responsible */
 got_quote(stock, date, time, price)
   char *stock, *date;
   int time, price;
   {
       register c = *stock;
       if(between(c, PrimLow, PrimHigh) || between(c, SecLow,
                                                SecHigh) ||
       (OldPrimLow && (between(c, OldPrimLow, OldPrimHigh) ||
                    ...))
                 /* bcast if I should disseminate this
                    quote */
                 address gaddr = pg_lookup(stock);
                 if(!addr_isnull(gaddr))
                     bcast(gaddr, NEW_QUOTE, ''%s: %s %d, %d'',
                          ..., 0);
            }
   }

/* Reconfigure after group membership changes */
ticker_mon(gv)
  groupview *gv;
  {
      /* Temporarily use both work decompositions */
      OldPrimLow = PrimLow; OldPrimHigh = PrimHigh; ...
      PrimLow = 'A' + 26/gv->gv_nmembers*my_rank(gv);
      PrimHigh = 'A' +
          26/gv->gv_nmembers*mod(1+my_rank(gv),
                                gv->gv_nmembers)-1;
      SecLow = .... etc;
      if(my_rank(gv) == 0) {
          /* Poll group members (including self) */
          bcast(tickgroup, POLL, '''', ALL, '''');
          /* All can switch over */
           bcast(tickgroup, SWITCH, '''', 0);
      }
  }

/* Send an empty reply; what counts is that I got
   the message */
poll(mp)
  message *mp;
  {
      reply(mp, '''');
  }

/* Switchover is completed. Stop monitoring stocks from
   old view */
switchover(mp)
  message *mp;
  {
      OldPrimLow = OldPrimHigh = ... = 0;
  }
```

as the routine monitoring the state of the ticker group. The method is slightly simplified for presentation. For example, it should be extended to deal with multiple failures by having a list of old mappings, instead of just one. It would also be desirable to cache group addresses.

The idea of virtual synchrony simplifies the switchover logic. Because the process doing the polling operation has already observed the new view, all processes that receive a poll message will also have seen it. Thus the replies they send need not carry any data. Because the sender waits for replies from everyone, the sender cannot meet our objective of executing the second-stage broadcast (the one to the **SWITCH** entry point) when all the processes have definitely received the new view.

The algorithm in Figure 11 cannot tolerate a second failure that occurs before it switches over the new work assignment. The problem is that the group could start a second reconfiguration while the first is still underway. The switchover message for the first failure would then be interpreted as if it applied to the second failure. One possibility is simply to accept the risk that a few quotes will be lost if two failures occur in rapid succession, although the odds are small. The alternative is to change **ticker_mon** to eliminate this problem by checking to see if the *view sequence number* for the group changed while the **POLL** was occurring. ISIS maintains view sequence numbers, which increment with each view change. If the number does change, the **SWITCH** message should not be sent. Some other task, which is also running **ticker_mon**, will be responsible for new reconfiguration.[12] Figure 12 shows a version that tolerates arbitrary sequences of failures.

```
ticker_mon(gv)
  groupview *gv;
  {
      int viewid = gv)gv_viewid;
      /* Record old values if reconfiguration is not already underway */
      if(OldPrimLow == 0)
          { OldPrimLow = PrimLow; OldPrimHigh = PrimHigh; ... }
      PrimLow = 'A' + 26/gv->gv_nmembers*my_rank(gv);
      PrimHigh = 'A' + 26/gv->gv_nmembers*mod(1+my_rank(gv), gv->gv_nmembers)-1;
      .... etc
      if(my_rank(gv) == 0) {
          bcast(tickgroup, POLL, '''', ALL, '''');
          /* Check to make sure view didn't change while waiting for replies */
          if(gv->gv_viewid == viewid
          bcast(tickgroup, SWITCH, '''', 0);
      }
  }
```

Figure 12. *A fault-tolerant version of the ticker_mon procedure.*

Although the logic behind this change is subtle, keep in mind that without ISIS, the same problem is nearly *impossible* to solve in fault-tolerant fashion. Also, much of the complexity stems from **ticker_mon**'s being a lightweight task that can be reentered while it is asleep in **bcast**, a potential problem in any system with lightweight tasks. ISIS may not make fault-tolerant reconfiguration easy, but it does make arriving at a concise, correct solution feasible.

The META Operating System

Mechanisms such as remote procedure calls and the ISIS tools act as a "glue" programmers can use to build distributed programs. With the ISIS Toolkit, programmers can concentrate on problems such as how certain data structures should be shared and how control should be distributed and ignore problems such as how failures can be detected consistently or how updates to a replicated data structure can be made atomic. In a formal sense, ISIS does not allow programmers to write more powerful programs, but it does make the task much easier, and the chances of the programmer's writing a correct fault-tolerant distributed program are much higher than if he were to use simple remote procedure calls.

Distributed *systems*, however, are not merely distributed programs. Applying the term *system* to a set of programs, distributed or centralized, implies that the interconnections between the programs are nontrivial. Moreover, a distributed system must deal with a complex and changing runtime environment. For example, consider the stock-brokerage system. We may want to monitor the news wires for keywords and have market-forecasting programs adapt to major world news events. When such an event occurs, the entire set of programs run may differ from the normal case. That is, a single external event can have sweeping implications that span most of the distributed system. To solve this kind of problem, especially if our system may have to deal with many such events in differing ways, we again need glue. Here, however, the glue permits us to do *programming in the large*.

In essence, a distributed system consists of a set of programs, which may be distributed ones, and a form of glue that controls and interconnects them. Typically, the system is mediated by the operating system. In current distributed systems, the varieties of glue consist of *network services* such as fileservers, electronic-mail servers, nameservers, lock managers, and even ticker services.

Unfortunately, network services are not everything that's needed. The programmer is still forced to worry about basic problems of how to monitor for an event, how to alert a program that an event has occurred, or how to ensure that the failure of a server won't cripple the system. Network services are lacking mechanisms analogous to those in the ISIS Toolkit, but they are oriented toward programming in the large. As a result, program interconnection in current distributed operating systems is unsophisticated at best. It is hardly surprising that current distributed systems provide little more than a collection of local operating systems supporting remote execution, load balancing, and transparent location of files.

Like ISIS, the META project provides a better glue. Our objective is to build a layer of operating-system-like software that will span many machines in a network: on the order of hundreds to thousands of workstations. META will not replace the underlying operating systems but instead will offer the higher-level glue to allow large fault-tolerant distributed applications to be easily interconnected and controlled.

Currently META consists of three major pieces:

1. A distributed, highly available filesystem built from standard network filesystems and that supports the standard NFS file-access protocols.
2. An event manager that allows programs to interact by using fault-tolerant events.
3. An event monitor that interprets policy rules written in a system-independent language.

META is in an early stage of development and experimentation. The structure will change and expand with time and experience.

The META File System

The filesystem is the pivotal component of a distributed system. Virtually all sharing of persistent data occurs through the filesystem, and much of the performance of a distributed system is determined by the performance of the filesystem. As a result, the majority of the current research taking place in distributed filesystems has focused on increasing performance.[15]

Performance is not the only property needed from a distributed filesystem. Also important is to ensure availability to key files; otherwise, the failure of a server can lock an application or the workstation itself. Key files include relatively static ones such as system-configuration files and dynamic ones such as log files and text files. Moreover, the distributed filesystem should provide the structure needed to interconnect perhaps hundreds of filesystems, including slow local ones, large shared repositories, and special-purpose filelike devices into a coherent whole. A distributed filesystem should also be easy to manage; current ones require too much effort on the part of the system administrators who partition disks schedule backup procedures.

The META File System consists of two parts: a *distributed control service* that uses file replication to give high availability and a set of *data repositories*. The control service implements both replication and the distributed filesystem abstractions needed to deal with large-scale file management. The repositories consist of commercially available file-servers that can also be used to store nonreplicated files.

The current prototype uses a simple file replication algorithm and stores files on NFS servers.[14] It is completely transparent to both clients and servers. Its structure is shown in Figure 13. The intermediate *agents* implement the META File System control service. They guarantee that all updates are ordered with respect to other updates, that all available replicas are written, and that the crash and later recovery of an NFS server makes the replicas on that server current. The replication incurs a cost, but UNIX caching hides the majority of it.

The META Event Manager

A filesystem lets programs share data, but it is not very useful for synchronization of programs. In order for programs to interact, they need a way to signal and await conditions, at a high level that can span large numbers of machines or programs.

For example, consider a utility called *PMake*, which is a distributed version of the UNIX *make* program. PMake needs to locate a set of machines that are lightly loaded, have the correct construction tools available, and have enough resources to complete a set of construction steps in a reasonable amount of time. Lacking META, PMake must solve this problem by talking with a name-server to locate a set of possible machines, a lock manager to tentatively allocate the machines, and *rstatd* to determine if the machine is lightly loaded. There simply isn't any easy way to decide if machines have the right tools or filesystems mounted. If these or other properties are taken into account, either an existing service would have to be expanded or a new one written.

The META Event Manager makes writing programs such as PMake easier. Like the META File System, the Event Manager is primarily a distributed control program that mediates between programs awaiting general events or needing resources into specific requests on existing services. PMake issues a description of its needs to the Event Manager and simply waits for the Event Manager to satisfy them. The Event Manager includes a generic "server" for new types of queries to be added easily. The architecture of the Event Manager is shown in Figure 14.

As seen by a client, the Event Manager has a set of *tables* and *functions* that represent, respectively, static and dynamic properties about the system. The client can query these tables with a simple procedural interface. At this level, the Event Manager resembles a *temporal database* with highly available tables.[15]

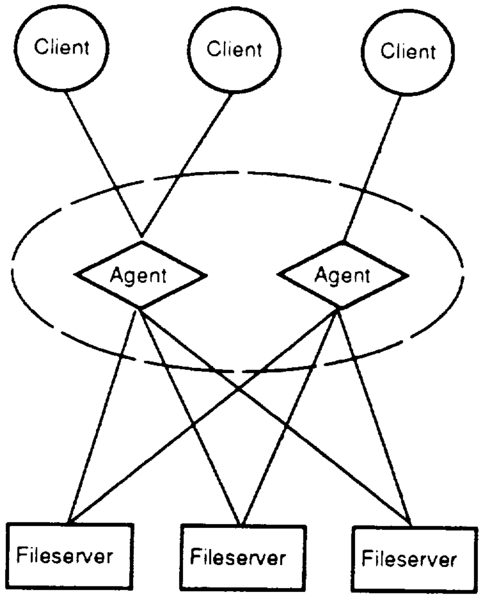


Figure 13. *META File System architecture. A substantially extended version should be operational in mid-1989.*

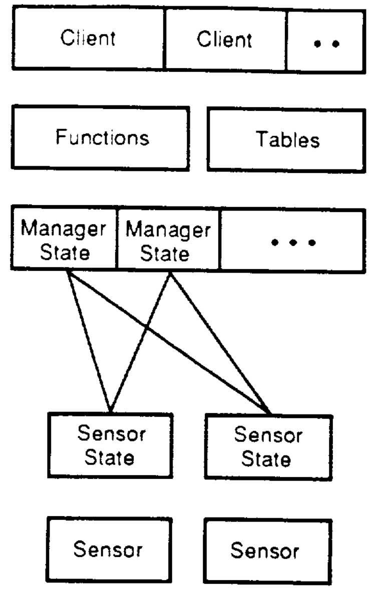


Figure 14. *META Event Manager architecture.*

Internally, the Event Manager maintains the tables and functions both as private tables and as queries to existing services. The Event Manager uses replication for fault tolerance. This manager can take advantage of any replication that the existing services already supply. For example, suppose the Event Manager maintains a function representing the temperature inside a reaction vessel. For this value to be available, *some* replication must exist; otherwise, the single failure of the only temperature sensor will make the function inaccessible.

Two methods are available for doing the replication: another temperature sensor or a pressure sensor and use of Boyle's law. By having programs read these values indirectly through the Event Manager, either physical value can be translated into the desired logical one by a single computation (supplied to us by the programmer who defines "temperatures").

New services can be added by a method similar to RPC *stub generation.* When creating a new service, the implementor writes an *interface* describing the information being sensed and the kind of sensor available (for example, edge sensitive or polled). A stub compiler generates a sensor stub that calls the sensor, a monitor stub the Event Manager uses to access the sensor, and location information that allows the Event Manager to bind to the sensor.

The META Event Monitor

PMake might want to allocate five Sun-4 workstations, each with a low load and 16 MB of memory; however, there may be other restrictions on the workstations chosen. For example, if a workstation is slated for maintenance in the next hour, or the network to which the workstation is attached will be used for several large file archives, the workstation should be considered temporarily unavailable. The programmer cannot know all the restrictions ahead of time, so the Event Manager denotes an abstract property of the workstation: its availability.

How will an event be generated? Rather than write a separate program monitoring each possible condition, the META Event Monitor keeps a simple *rule base* specifying the conditions that lead to a workstation's unavailability. Here, one rule might be:

```
DURING StartPM(machine) - 1 hour TO EndPM(machine)
      THEN Unavailable(machine)
```

The META Event Monitor is not meant only for monitoring network conditions. It is useful for specifying any *policy rules* that can be translated into basic actions that other programs will follow. A more elaborate example is a hospital system containing several distributed programs, such as programs that locate a doctor and send emergency messages, sensor systems that monitor patients, programs that schedule operating rooms, and systems that prescribe drugs. Policy rules can tie these programs together—for example, to alert a doctor if a patient has an adverse reaction to a drug, to assemble the necessary resources for an emergency admission of a patient, or to locate a replacement doctor if the primary one is not available. The policy rules can be altered as the resources of the hospital change, but the programs supplying the mechanisms need not change.

Using the Event Monitor offers several advantages. It permits policy rules to be separated from programs and specified in an explicit and concise manner and builds a special-purpose program to implement each rule. The rules can evolve without the necessity of extensive system changes or reprogramming.

The Event Monitor itself is a distributed program that behaves somewhat like an expert system. It maintains a set of rules, which it continuously and concurrently evaluates. The rules are written in a realtime version of interval logic[16] and are evaluated against the META Event Manger's tables and functions.

Availability

ISIS is publically available in the United States and subject to some minor export restrictions in most other countries. Commercial support for ISIS is available from ISIS Distributed Systems, Inc. Source is provided with the system, which can currently be used to interconnect Sun, HP, DEC, Apollo, and Gould computer systems running variants of Berkeley UNIX. A MACH port has been completed, and ports to AIX and possibly VMS, as well as interfaces to the toolkit from other languages, are planned. The META Operating System is still under development; plans to distribute it have not yet been established.

Acknowledgments

Listing all the contributors to the ISIS Toolkit and the META Operating System effort is not feasible. Among all these people, however, special recognition is due to Tommy Joseph, Ken Kane, Frank Schmuck, and Mark Wood, all of whom have made invaluable contributions to the project. This work was funded by the Department of Defense Advanced Research Projects Agency under grant N00140-87-C-8904, October 1988.

Footnotes and References

1. Carriero, N.; and Gelertner D.; "The S/Net's Linda Kernel," *ACM Transactions on Computer Systems.* May 1986, pp. 110-129.
2. Hoare, C.A.R, "Communicating Sequential Processes," *Commun. ACM*, August 1987, pp. 666-777.
3. Liskov, B., "Implementation of Argus," *Proc. 11th ACM Symposium on Operating Systems Principles*, November 1987, pp. 111-123.
4. Spector, A.; Pausch, R.; and Bruell, R.; "Camelot: A Flexible, Distributed Transaction Processing System," *Proc. IEEE Compcon 88*, San Francisco, CA, February 1988, pp. 432-437.
5. ISIS is named after the Egyptian goddess who mummified the remains of Osiris after his defeat in a battle with Set, bringing him back to life as the ruler of the Underworld and setting the stage for the eventual defeat of Set in a battle with their son Horus.
6. Birman, K.P.; Joseph, T.; Kane, K.; and Schmuck, F.; *The ISIS Programming Manual and User's Guide*, Department of Computer Science, Cornell University, June 1988.
7. ISIS does handle the special case of a single machine that gets partitioned off from the others. In this case, ISIS treats the partitioned system as if it had crashed, and it will have to rejoin the operational sites when communication is restored. Tools are available to make this process as painless as possible.
8. Our use of the term *broadcast* should not be confused with the broadcast feature that some LAN communication devices provide. An ISIS broadcast is just a message transmission to a process group. Although a hardware broadcast might be useful for optimizing the delivery of an ISIS broadcast, ISIS normally uses point-to-point messages because hardware broadcasts might not work over bridges or gateways.
9. Bernstein, P.; Goodman, N.; and Hadzilacous, V.; *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Series in Computer Science, 1987.
10. Ibid.
11. Birman, K.P.; and Joseph, T.; "Reliable Communication in an Unreliable Environment," *ACM Transactions on Computer Systems*, February 1987, pp. 147-76. Birman, K.P.; and Joseph, T.; "Exploiting Replication," *Arctic '88: An Advanced Course on Distributed Systems*, Addison-Wesley, 1989.
12. Notice that a second task running **ticker_mon** can be started before the first one terminates. Worse, the view can change while a **ticker_mon** task is asleep in the **POLL** broadcast. The programmer who codes an algorithm like this needs to understand enough about lightweight tasks to realize that this is a potential problem. In this case, the **gv->gv_flag** is checked to make sure the view is still valid (current) after **POLL** terminates.
13. Howard, John H.; Kazar, Michael L.; Menees, Sherri G.; Nichols, David A.; Satyanarayanan, M.; Sidebotham, Robert N.; and West, Michael J.; "Scale and Performance in a Distributed Filesystem," *ACM Transactions on Computer Systems.* February 1988, pp. 51-82.
   • Nelson, Michael N.; Welch, Brent B.; and Ousterhout, John K.; "Caching in the Sprite Network Filesystem," *ACM Transactions on Computer Systems*, February 1988, pp. 134-154.
14. Marzullo, Keith; and Schmuck, Frank; "Supplying High Availability with a Standard Network Filesystem," *Proceedings of the Eighth DCS Conference*, June 1988.
   • *Networking on the Sun Workstation*, revision B, February 1986. Sun Microsystems, Inc., 2550 Garcia Ave., Mountain View, CA 94043.
15. Snodgrass, Richard, "A Relational Approach to Monitoring Complex Systems," *ACM Transactions on Computer Systems.* May 1988, pp. 157-196.
16. Schwartz, R. L.; Melliar-Smith, P. M.; and Vogt, F. H.; "An Interval Logic for Higher-Level Temporal Reasoning," *Proceedings of the Second PODC.* August 1983, pp. 173-186.